Visualization of Diffusion Limited Aggregation with OpenGL

Clayton Rayment[1]

Abstract
An efficient single-threaded application for performing Diffusion Limited Aggregation simulations, along with a companion OpenGL visualizer using the GLUT GL Utility Toolkit. Particle-aggregator collisions are detected in $O(n \log n)$ time complexity, however this could be improved with the implementation of tree-based collision detection. This code is available here: `https://github.com/simharry3/DLA`

Keywords
DLA — Visualization — OpenGL

Contents

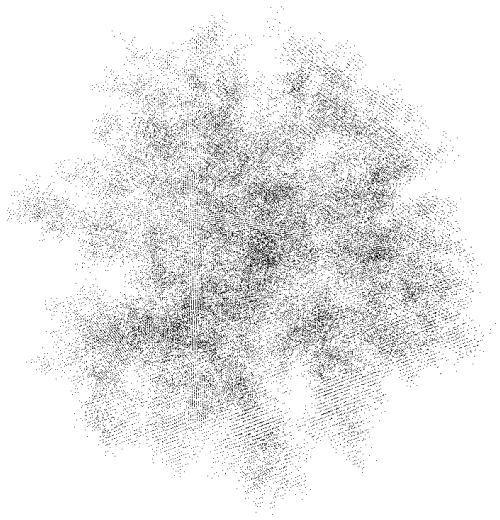

Figure 1. Aggregate structure resulting from a DLA simulation using 50,000 particles. Aggregate structure shown contains 45,007 particles.

Introduction

Diffusion Limited Aggregation is a simulation technique consisting of two basic particle types. Aggregator particles make up the first group of particles. An Aggregator particle does not move, and only interacts with the simulation when an Active Particle collides with it. The second group of particles, Active Particles, undergo a random walk at each timestep of the simulation. Should an Active Particle collide with an aggregator particle, the active particle will cease to be active, and become an aggregator particle in the location from where it collided. In this manner, an aggregate structure will grow as the simulation progresses. An example of such a structure can be seen in Figure 1.

1. Implementation

Classes

Particle

The `Particle` class is the most basic object of the simulation. The particle class defines only the location of the particle, the particles morton code, and functions to retreive/set these values. During the simulation, a particle will move randomly in one of eight possible directions by adding a random integer value [-1, 1] to each component of the particles position. Particles also contain a function which encodes the position into a 30-bit morton code, which is used to accelerate aggregation detection.

Universe

The `Universe` class is the overarching structure which contains all simulation operations. `Universe` contains an `std::vector` of all aggregated particles, and an `std::list` of all active particles. During each step of the simulation, the active particle list is iterated through, and each particle is moved, checking for collisions with the aggregate particles. For simplicity, collsions between active particles cannot occur, and the two particles will simply pass through eachother. Boundary conditions for the universe are enforced by not allowing a particle to travel beyond the simulation bounds. Should a particle be assigned a random motion that would take it outside of the simulation bounds, the component that would exceede the simulation bounds is capped at the bound.

Fast Aggregate Lookup

To speed up collision detection with the aggregate particles, each aggregate particle is assigned a Morton code based on the position of the particle. The `std::vector` container which holds all aggregator particles is then sorted using the aggregator particle's morton code.

During the collision check step, rather than search the entire aggregator particle vector looking for a particle that matches the position of the desired movement, Morton encoding allows the vector to sorted in a deterministic way, and searched using `std::binary_serch`. This reduces the lookup time from $O(n)$ to $O(\log n)$.

Visualizer

Visualization is done using OpenGL. To increase performance, the visualizer is run in a separate thread using Pthreads. When the `Universe::renderUniverse()` method is called, the visualizer is started, and passed a pointer to the Universe class that called the visualizer.

freeGLUT

freeGLUT is an open source alternative to the GL Utility Toolkit (GLUT), which is a library for C/C++, FORTRAN, and Ada. Writing a visualizer with freeGLUT consists of writing several functions and registering them with freeGLUT so that they can be executed during certain events. These include:

- keyboard input
- mouse input
- window resize
- program idle

2. Operation

Aggregator Generation

An aggregator file is a space-seperated value list containing only numbers in the following format:

```
<X> <Y> <Z>
```

Each line in the file represents a new aggregator particle in the system. Several sample aggregate structures are located in the `<dir>/samples` directory. Should the user not specify an imput aggregator file a default aggregatior will be placed at the center of the simulation.

Build Simulation

Cmake files are included to automate the build process. From your build directory simply run

```
cmake <dir>
```

where `<dir>` is the path to the main `DLA/` directory which contains `CMakeLists.txt`. Then, from your build directory, simply run:

```
make
```

You will now find the executables in the `<dir>/bin/` directory.

Run Simulation

Once the project has been built, the simulation can be run from the command line using:

```
<dir>/bin/simulation <Size> <Particles>
```

A third, optional argument may be used to specify the location of the input aggregator file, otherwise the default aggregation will be used.

Visualizer

Several options are available to the user from the visualizer, as displayed at the bottom of the visualization window:

Esc Close the simulation window and end the simulation.

F1 Toggle render mode between fast and fancy.
(Default: fast)

F2 Toggle bounding box to cover the simulation space.
(Default: off)

F3 Toggle rotation of camera around simulation.
(Default: off)

F4 Toggle visibility of active particles in simulation.
(Default: on)

An example of the visualizer window is shown in Figure 2.

Acknowledgments

Morton Encoding code taken from [1]

References

[1] T. Karras. Maximizing parallelism in the construction of bvhs, octrees, and k-d trees. *High Performance Graphics*, 2012.

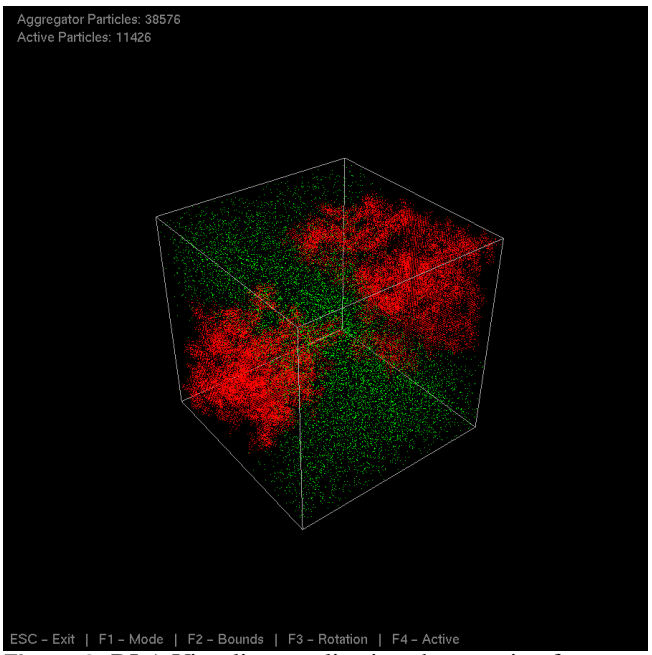

Figure 2. DLA Visualizer application shown using fast graphics, bounding box, and active particles visible (green).